DESENVOLVIMENTO DA PROPOSTA DE SOLUÇÃO

Em conformidade com os requisitos e casos de uso definidos na proposta de solução, foi desenvolvido o protótipo de sistema. O processo de desenvolvimento iniciou-se pela preparação do ambiente de desenvolvimento e instalação de todas as dependências necessárias. Posteriormente, foi implementada a tela de gerenciamento do sistema, relacionada ao caso de uso Gerenciar Sistema. Após esta fase inicial, os cadastros de *marketplace* e vendedor foram desenvolvidos. Somente após a conclusão destes cadastros que o módulo do sistema responsável pela combinação de categorias e atributos foi desenvolvido. Por fim, foi desenvolvida a API REST para consulta das combinações criadas no sistema. Conforme definido na proposta, o protótipo foi desenvolvido com a linguagem de programação Java, *framework Spring Boot* e utilizando a arquitetura MVC.

Uma empresa responsável por uma plataforma de *marketplace* própria, localizada no estado do Rio Grande do Sul, foi contatada durante o desenvolvimento do projeto. Esta empresa, mesmo que de forma sigilosa, forneceu dados relativos à sua taxonomia e disponibilizou sua API de *marketplace* para eventuais testes. Após a análise das informações coletadas, algumas mudanças pontuais foram realizadas na arquitetura do protótipo, para atender a necessidade real desta empresa, que, no presente trabalho, será referenciada como “empresa XY”.

ARQUITETURA IMPLEMENTADA (coloquei esse nome porque já tem um capitulo chamado arquitetura do sistema)

A arquitetura MVC e o padrão de projetos *composite* foram a base arquitetural do protótipo desenvolvido. Entretanto, a implementação do padrão *composite* sofreu uma alteração após a análise da estrutura taxonômica do catálogo de produtos da empresa XY.

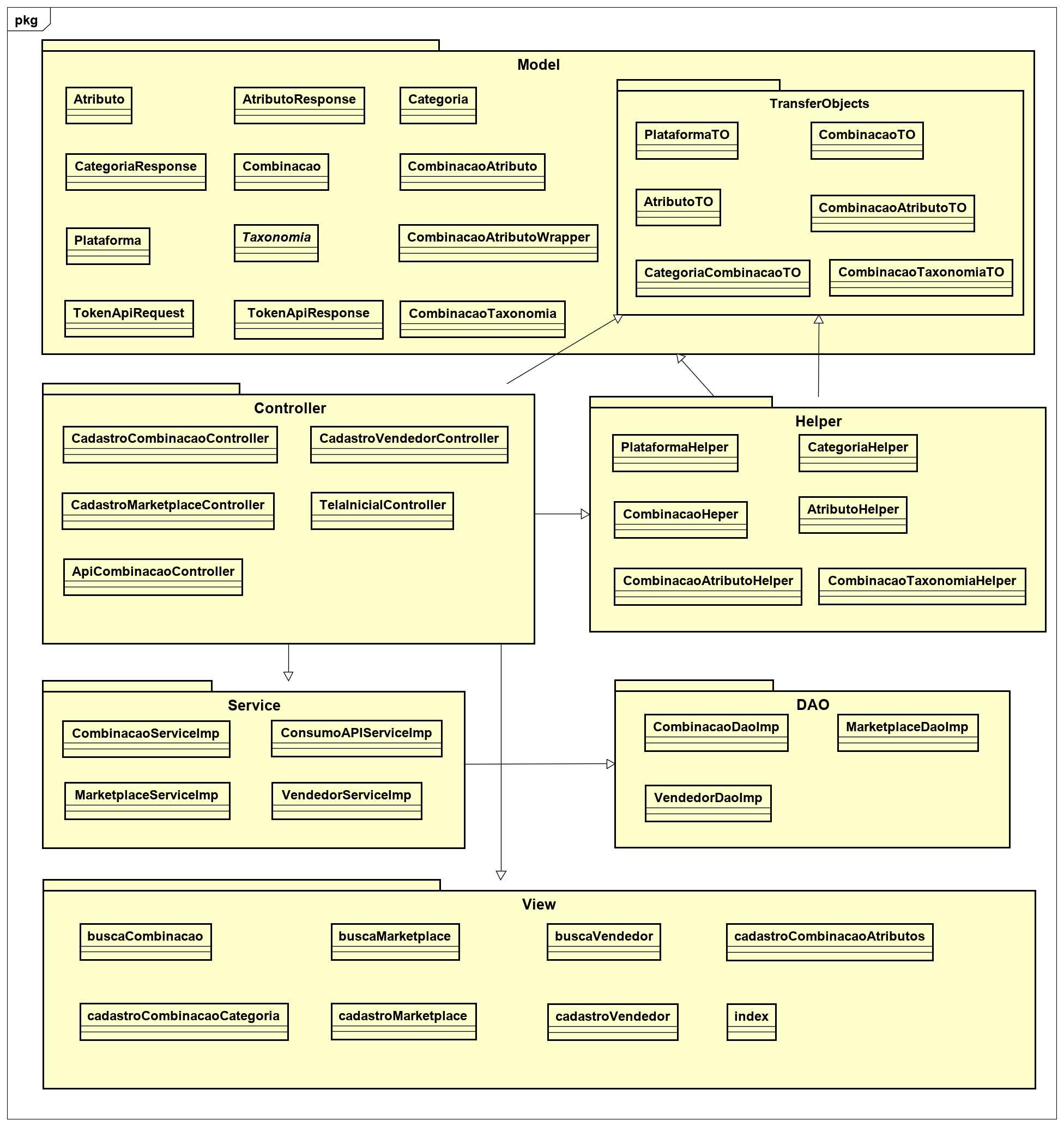
Inicialmente a implementação do padrão *composite* estava prevista para todo o processo de combinação de taxonomias, como pode ser visto na Figura 5555. Contudo, a estrutura do catálogo da empresa XY é muito restrita, definida obrigatoriamente em três níveis de categoria. Estes níveis foram denominados “linha”, “família” e “grupo”, sendo “grupo” o nível mais específico da estrutura. Levando em consideração este modelo de taxonomia, foram criadas as classes Combinacao e CombinacaoTO representando a estrutura da empresa XY. Esta modificação contribuiu para a legibilidade do código, deixando-o menos complexo e contribuindo com a manutenibilidade do sistema. Apesar da mudança ocorrida, o padrão *composite* ainda foi bastante utilizado, principalmente no processo de consumo de APIs e persistência dos dados relativos a categorias e atributos dos vendedores e marketplaces.

No decorrer do desenvolvimento do protótipo, para garantir uma melhor organização e reaproveitamento de código, foi optado pela utilização do padrão *Data Access Object* (DAO). Conforme Engholm Junior (2013), o padrão DAO é utilizado para o encapsulamento da lógica de acesso ao banco de dados, promovendo a reutilização do código uma vez que não há o acoplamento à camada de negócios. Ainda segundo o autor, juntamente com este padrão pode ser utilizado o padrão *TransferObject* (TO) para trocas de dados entre componentes do sistema. No protótipo desenvolvido este padrão também foi utilizado, entretanto, não em sua totalidade.

Na implementação foram definidos dois tipos de classes na camada *Model*. As classes de negócio e as classes TO. As classes de negócio são utilizadas internamente pelo sistema e transacionam dados entre as camadas *Controller*, *Service* e DAO. Já as classes TO são utilizadas para transacionar dados apenas entre a camada de *View* e a camada *Controller*. Foi optado pelo uso do padrão TO devido à complexidade que o sistema adquiriu durante seu desenvolvimento. Este padrão foi implementado após grande parte do desenvolvimento já ter sido realizado. Por este motivo, algumas classes ainda precisam de adequação, pois transacionam apenas objetos TO entre as camadas.

Para conversão dos objetos TO em objetos de negócio foram criadas classes com métodos estáticos localizadas em uma camada chamada *Helper*. A Figura 22 representa as camadas do sistema, bem como as classes e arquivos HTML referentes às interfaces gráficas.

Figura 22 – Diagrama de Pacotes



Fonte: Elaborado pelo autor (2019)

GERENCIAR SISTEMA

Com o objetivo de possibilitar a navegação do usuário entre as funcionalidades do sistema, foi desenvolvida a tela de navegação inicial. Esta foi a primeira tela desenvolvida no protótipo e a partir do seu desenvolvimento estabeleceu-se a estrutura de todos os componentes gráficos do sistema. Um modelo desta tela pode ser analisado na Figura 666.

Figura 777 – Tela de gerenciamento do sistema



Fonte: Elaborado pelo autor (2019)

Além de HTML e CSS, foram utilizadas diversas outras tecnologias de desenvolvimento de interfaces gráficas web, são elas: *Thymeleaf*, B*ootstrap* e *Jquery*.

*Thymeleaf(1)* é um motor de *templates* *server-side* utilizado em conjunto com a linguagem de programação Java. Os modelos HTML escritos com *Thymeleaf* ainda parecem e funcionam como HTML, permitindo que *o de design* dos artefatos possam ser modificados normalmente. O *Boostrap(2)* por sua vez, se trata de uma biblioteca de componentes para design de interfaces gráficas desenvolvida com HTML, CSS e *JavaScript*. Esta biblioteca abstrai o uso de código CSS e Javascript, além disso, os componentes existentes melhoram a experiência do usuário através de *templates* amigáveis e responsivos. A Figura 777 mostra um trecho de código utilizando *Thymeleaf* juntamente com *Boostrap*.

Figura 777 – Exemplo *Thymeleaf* e *Boostrap*



Fonte: Elaborado pelo autor (2019)

*Jquery(3)* é uma biblioteca *JavaScript* muito versátil e simples, que facilita a utilização de animações, manipulação de eventos e manipulação de documentos HTML em tempo real. No protótipo, o Jquery foi utilizado em todas as interfaces, principalmente na tela de combinação de categorias e atributos em conjunto com o *Boostrap*, como mostra a Figura 888

Figura 888 – Exemplo de código com utilização de *Jquery*



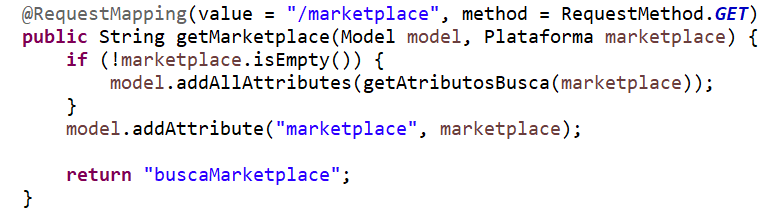
Fonte: Elaborado pelo autor (2019)

CADASTRO DE MARKETPLACE E VENDEDOR

O cadastro de *marketplace* e vendedor são pré-requisitos para que as combinações de categorias e atributos sejam criadas. Apesar do cadastro de vendedores e do cadastro de marketplaces possuírem estruturas separadas tanto na aplicação, como no banco de dados, o desenvolvimento a nível de código e os processos utilizados são os mesmos. Até mesmo graficamente estes dois módulos do sistema são muito similares. Por estes motivos os exemplos relacionados a seguir levarão em consideração o cadastro do *marketplaces*.

O processo de cadastro de um marketeplace é iniciado pelo acesso à tela de cadastro de marketplaces. Na tela inicial, quando o usuário clica no botão “Cadastrar Marketplace”, a aplicação redireciona-o para a URL “/marketplace”. Esta URL está mapeada na classe chamada “CadastroMarketplaceController”, localizada na camada *controller*. Dentro desta classe, o método “getMarketplace” adiciona um objeto do tipo “Plataforma” como atributo do objeto “*model*” proveniente da classe nativa do Spring de mesmo nome. Por fim, o método retorna a *view* de busca de *marketplace*. A Figura 999 destaca o trecho do código descrito acima.

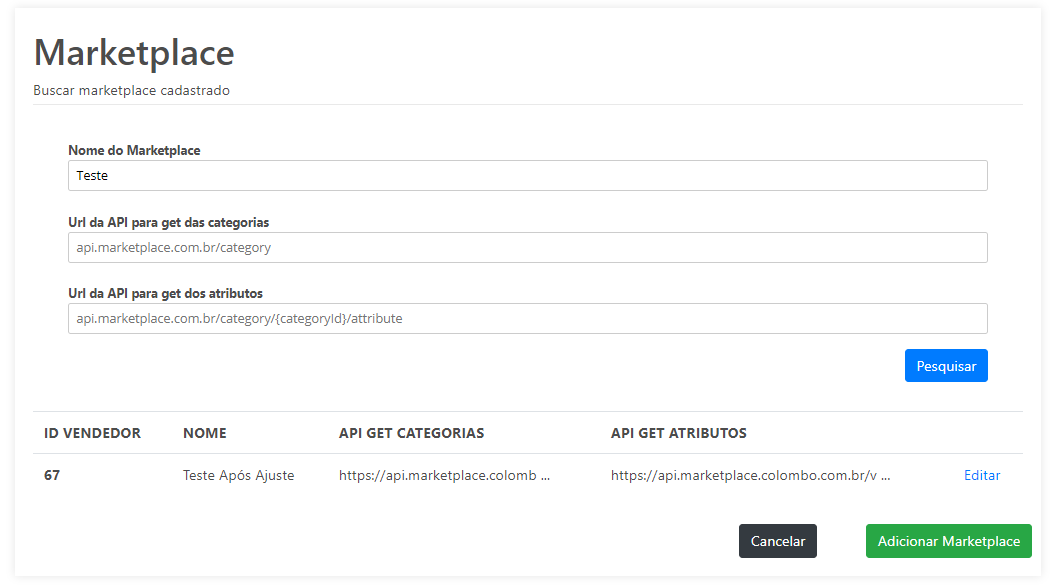
Figura 999 – CadastroMarketplaceController



Fonte: Elaborado pelo autor (2019)

Na tela de busca de *marketplace* retornada, o usuário terá a opção de buscar um marketplace existente para edição ou então poderá optar pelo cadastro de um novo marketplace através do botão “Adicionar Marketplace”. Ao pesquisar, o usuário é direcionado para o método “getMarketplace” já citado anteriormente, contudo, o objeto do tipo “Plataforma” estará instanciado com os dados preenchidos nos campos disponíveis na tela. Caso isso ocorra, a aplicação irá buscar os marketplaces existentes com base nos dados informados e então irá exibi-los na tela, conforme a Figura 1000.

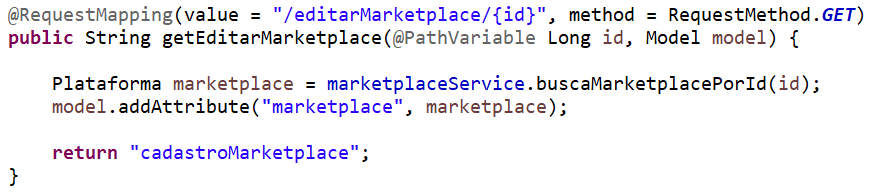
Figura 1000– Busca de *marketplaces*



Fonte: Elaborado pelo autor (2019)

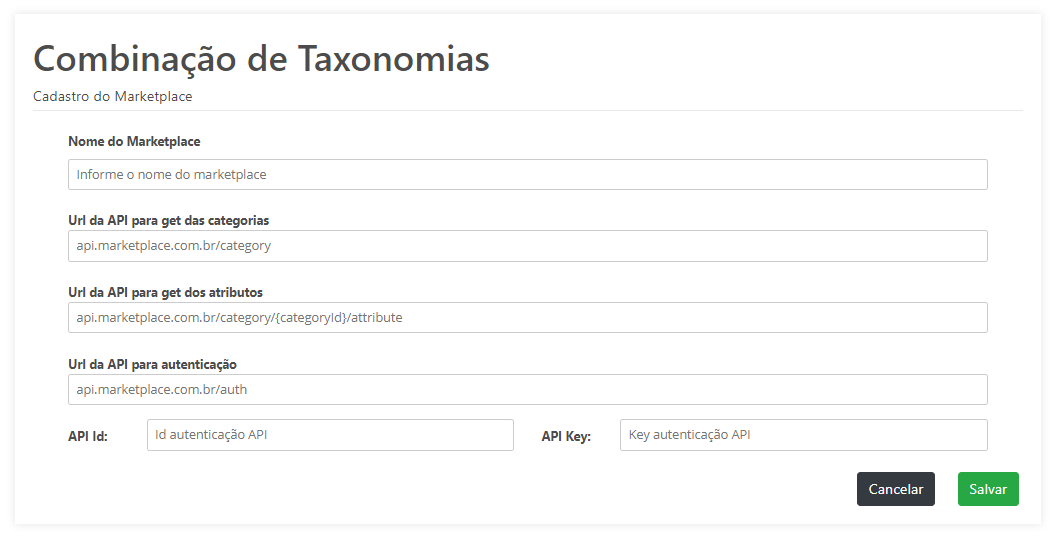
Caso o usuário opte pela edição de um *marketplace* existente, deverá clicar sob o *link* “Editar”. Este *link* direciona o usuário para a URL “/editarMarketplace/{id}”, atribuindo o campo id do *marketplace* selecionado como parâmetro. Então, o método do *controller* responsável por mapear essa URL irá buscar o marketplace em questão através do id, adicioná-lo como atributo do objeto model e retornará a *view* cadastroMarketplace, conforme a Figura 1111. Como a *view* de edição e cadastro de *marketplace* é a mesma, o processo que ocorre ao selecionar o botão “Adicionar Marketplace” é muito similar, a única diferença é que nenhum *marketplace* será buscado e adicionado ao *model*. A tela de cadastro e alteração de *marketplace* pode ser analisada na Figura 12222.

Figura 1111– Método getEditarMarketplace



Fonte: Elaborado pelo autor (2019)

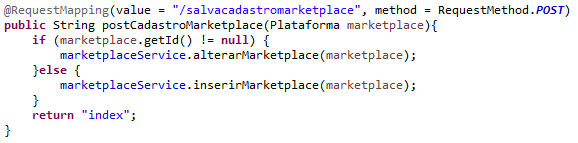
Figura 12222– Tela de cadastro de *marketplace*



Fonte: Elaborado pelo autor (2019)

O processo de cadastro é o mesmo para *marketplaces* e vendedores. Inicia-se pelo preenchimento dos campos do formulário e é concluído no momento em que o usuário clica no botão “Salvar”. Ao clicar neste botão, tanto na edição, quanto no cadastro de um novo *marketplace*, o usuário é direcionado para a URL “/salvacadastromarketplace” mapeada no *controller* “CadastroMarketplaceController” . O método responsável por esse mapeamento faz a distinção entre uma edição e um novo cadastro, utilizando dois processos distintos para salvar os dados. Este método pode ser visto na Figura 13333.

Figura 13333– Método postCadastroMarketplace



Fonte: Elaborado pelo autor (2019)

Outro detalhe que pode ser analisado na Figura 13333 é a utilização da interface “MarketplaceService” para acesso às classes da camada de service. Isso ocorre porque o *Spring Boot,* framework utilizado para desenvolvimento*,* exige que a aplicação seja separada em camadas bem definidas. Para atender tal necessidade, cada classe da camada de serviço e DAO possui sua respectiva interface. Estas interfaces são utilizadas como meio de acesso às classes de cada camada.

O método “inserirMarketplace” pertencente a camada de serviço possui três etapas: Persistência dos dados cadastrais do marketplace, consulta à API cadastrada para persistência dos dados relativos à estrutura de categorias e, por fim, consulta recursiva à API cadastrada para persistência dos dados relativos à estrutura de atributos. Estas etapas são pré-requisitos para que o cadastro de uma plataforma de marketplace ou vendedor seja concluído. Somente após este processo que as combinações de categorias e atributos poderão ser realizadas.

Para realizar o consumo das APIs cadastradas foram utilizados recursos específicos do *Spring Boot*. O principal deles é a classe nativa do Spring chamada *RestTemplate*. Esta classe possui uma série de métodos de fácil implementação para tratar requisições com APIs REST. No trecho de código da Figura 1444 são exibidos os três métodos da classe “ConsumoApiServiceImp”. Esta classe é responsável por todas as consultas em APIs que o sistema realiza durante o cadastro de vendedores e *marketplaces*. O primeiro método, “getAutenticacao”, é responsável por gerar o *token* de autenticação na API do seller ou marketplace. O segundo, “getCategorias”, realiza uma requisição GET na API utilizando o *token* gerado como parâmetro. Para consulta dos atributos por categoria o processo é similar, a diferença é que o código da categoria também é passado como parâmetro. Em todos os métodos a classe *RestTemplate* é utilizada.

Figura 1444 – Classe ConsumoApiServiceImp

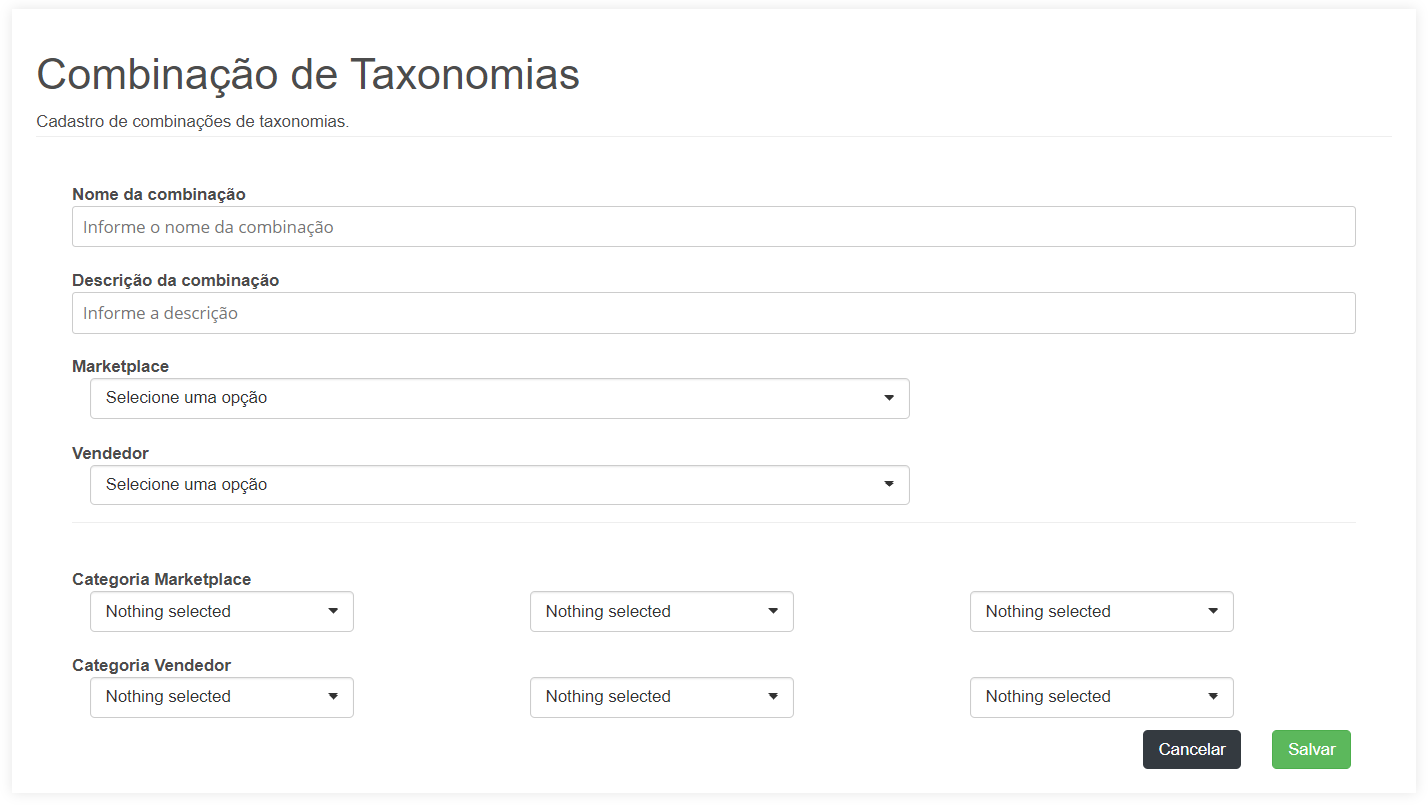


Fonte: Elaborado pelo autor (2019)

COMBINAÇÃO DE TAXONOMIAS

A combinação de categorias e atributos que compõe a taxonomia dos *marketplaces* e vendedores é o principal objetivo do sistema. Este processo só pode ser iniciado após o cadastro de pelo menos um marketplace e um vendedor.

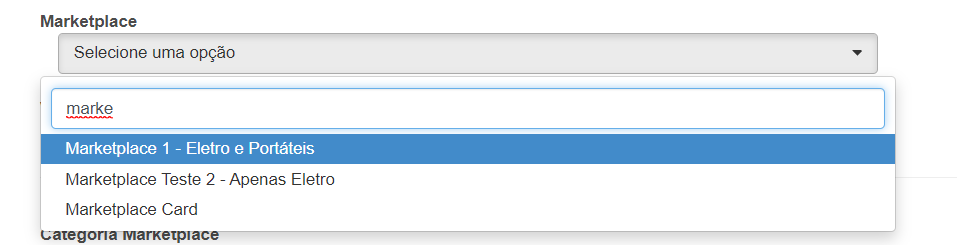
O primeiro passo para criação de uma nova combinação é o acesso a tela de cadastro. Para tal, o usuário deve clicar no botão “Criar Nova Combinação”, presente na tela inicial do sistema. Este botão redireciona o usuário para a URL “/cadastrocombinacaocategoria”, que está mapeada no “CadastroCombinacaoController”. Este *controller* busca e atribui para a view todos os marketplaces e vendedores cadastrados no sistema, além de um objeto do tipo “CombinacaoTO”. A Figura 1555 mostra a tela de cadastro de combinação de categorias.

Figura 1555 – Tela de cadastro de combinação de categorias

Fonte: Elaborado pelo autor (2019)

Para melhorar a experiência do usuário foram utilizados diversos recursos provenientes do Bootstrap e Jquery. Nas telas de combinação de categorias e atributos em especial, o componente *selectpicker* foi amplamente aplicado. Este componente possibilita que dados contidos em tags HTML do tipo *select* sejam pesquisados em tempo real, facilitando e agilizando o processo de combinação. A correta implementação deste componente exigiu um tempo considerável de desenvolvimento, pois, as versões do Bootstrap e Jquery inicialmente utilizadas não eram compatíveis com esta tecnologia. A figura 1666 demonstra a utilização deste componente no código e também como ele é renderizado ao usuário.

Figura 1666 – Componente *selectpicker*

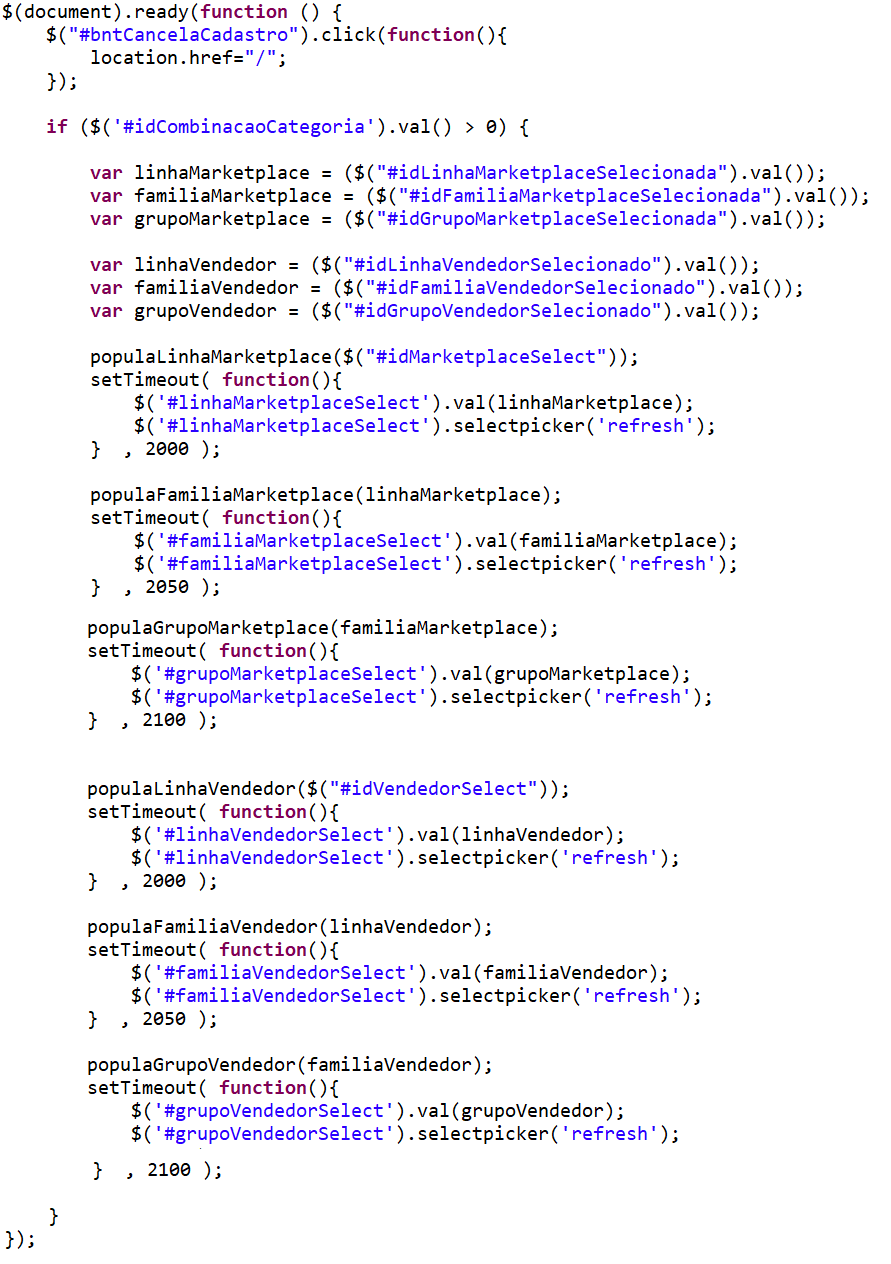


Fonte: Elaborado pelo autor (2019)

Grande parte das funcionalidades da tela de cadastro de combinação de categorias não são executadas ao carregar a tela inicialmente. Diversas funcionalidades são executadas conforme a interação do usuário na tela através de código JavaScript e JQuery. Ao selecionar um *marketplace* o sistema, em tempo de execução, busca o primeiro nível de categoria cadastrada levando em consideração o código do *marketplace* selecionado. Desta forma, apenas as categorias correspondentes ao *marketplace* selecionado serão exibidas, melhorando a experiência do usuário e facilitando o processo de cadastro. O mesmo também ocorre quando um vendedor é selecionado.

Na seleção de categorias o sistema também busca facilitar o cadastro. Sempre que uma categoria é escolhida, funções JavaScript são executadas buscando as categorias filhas tendo como base a categoria pai escolhida. A implementação destas funcionalidades não é complexa, contudo, problemas foram encontrados ao desenvolver a edição de combinações de categorias que também utiliza esta tela. Isso porque, para auxiliar ao máximo o usuário, foi definido que a combinação de categorias existente deveria ser exibida ao usuário ao acessar a tela. Desta forma, o mesmo poderia realizar apenas as alterações necessárias levando em consideração o que já havia sido cadastrado anteriormente. Para que este comportamento ocorresse, todas aquelas funções JavaScript executadas de forma assíncrona conforme interação do usuário deveriam ser executadas ao renderizar a página. Entretanto, foi constatado durante os testes que a aplicação Java estava sendo executada muito rapidamente, antes mesmo das funções JavaScript serem carregadas. A alternativa encontrada foi a utilização da função “setTimeout” provinda do JQuery para atrasar a execução das funções em alguns milésimos de segundo. Na Figura 1777 pode ser analisado o trecho de código JavaScript mencionado.

Figura 1666 – JavaScript executado ao editar combinação

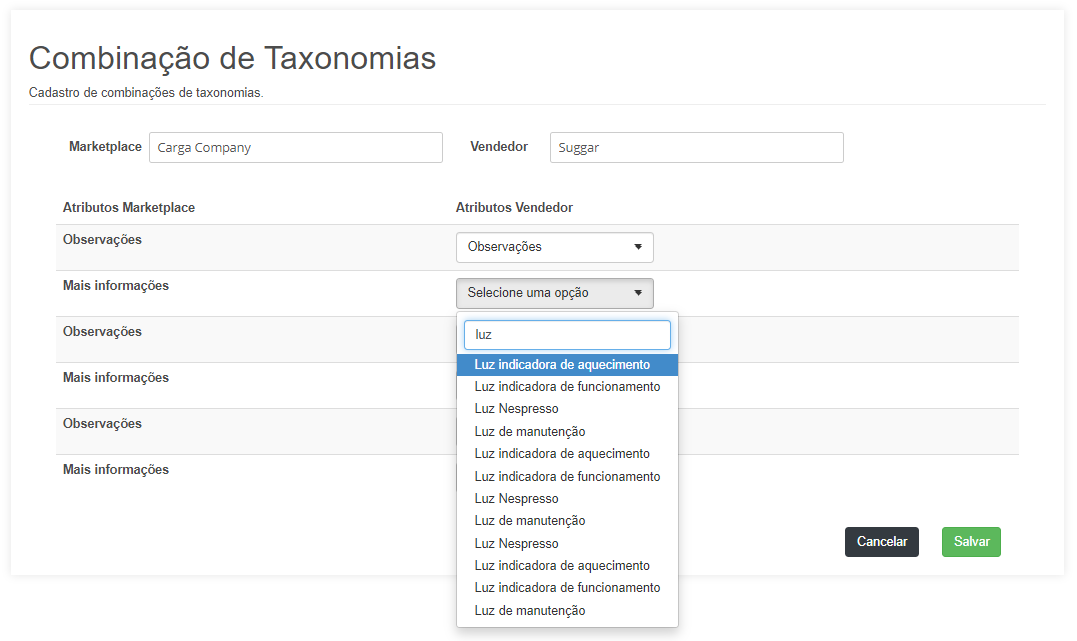


Fonte: Elaborado pelo autor (2019)

Quanto o usuário finaliza a combinação de categorias e clica no botão “Salvar”, as informações são persistidas no bando de dados e o usuário é direcionado a tela de combinação de atributos. Com base no *marketplace*, vendedor e categorias selecionadas na combinação de categorias a aplicação irá buscar todos os atributos existentes, separando-os em atributos de *marketplace* e vendedor. Após relacionar os atributos e clicar em salvar, os dados são persistidos no banco de dados. Um modelo da tela de combinação de atributos pode ser analisado a Figura 1777.

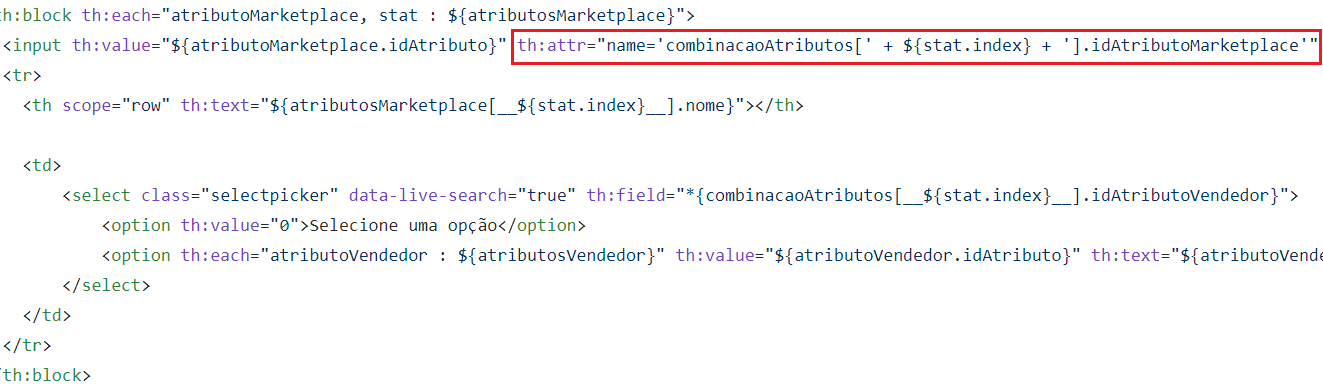
Esta etapa do desenvolvimento do protótipo foi a que mais demandou tempo em todo o projeto. O desenvolvimento desta tela foi complexo devido à necessidade de criar recursivamente os *inputs* HTML dos atributos do *marketplace* em uma tabela e, a cada iteração recursiva, criar um componente *select* com todos os atributos do vendedor. Conforme a documentação disponibilizada pelo próprio Thymeleaf(8), para adicionar *inputs* HTML ao código deve-se utilizar a o comando “th:field”. Contudo, para a situação descrita acima, este comando não funciona retornando um erro ao carregar página. Para contornar este problema foi utilizado o comando “th:attr”, normalmente utilizado para definir um atributo *value* em inpunts HTML, para definição do atributo *name*. Mais detalhes da solução adotada podem ser visualizados na Figura 1888.

Figura 1777– Tela de combinação de atributos



Fonte: Elaborado pelo autor (2019)

Figura 1888– View cadastroCombinacaoAtributos.html



Fonte: Elaborado pelo autor (2019)

CONSULTA DE COMBINAÇÕES

Foram criadas duas formas de visualização das combinações criadas. A consulta visual e a consulta via API. A consulta visual pode ser realizada pesquisando e selecionando a combinação existente na tela de consulta de combinações. Já a consulta via API deve ser realizada consumindo a API desenvolvida no projeto.

O processo para consulta visual de combinações inicia-se com o acesso a tela de consulta de combinações. Posteriormente, o usuário deve buscar a combinação desejada através dos filtros da tela de busca. Com base nas informações preenchidas, a aplicação realizará uma consulta no banco de dados e retornará as combinações encontradas, conforme a figura 1999.

Figura 1999– Tela para consulta de combinações

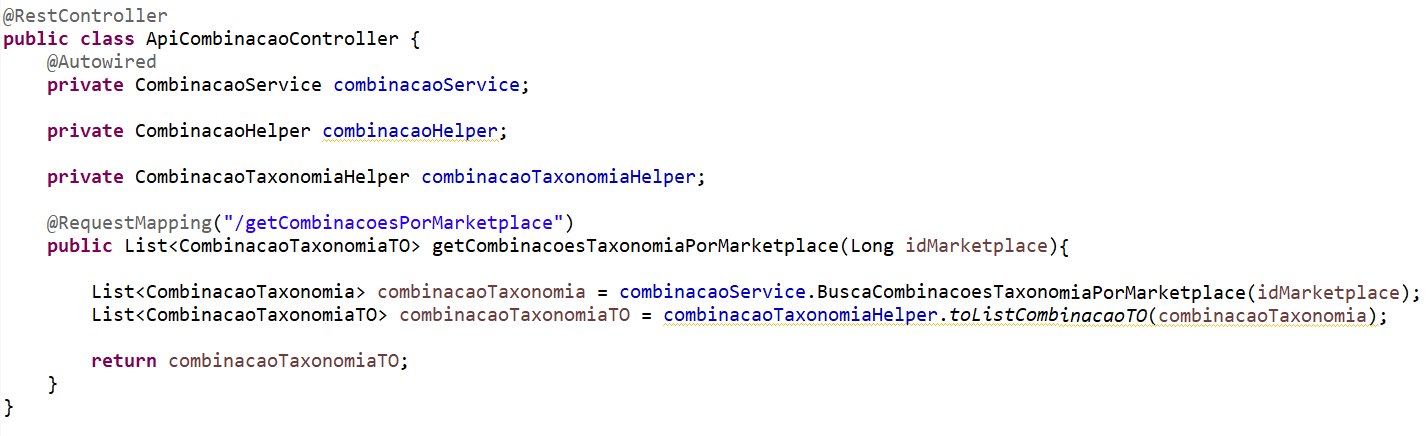


Fonte: Elaborado pelo autor (2019)

Para construção da API de consulta de combinações foram utilizados recursos do *Spring Boot* e da própria linguagem de programação Java. A anotação @RestController provida pelo framework Spring Boot, foi o recurso mais útil desta etapa do desenvolvimento. Esta anotação foi introduzida no Spring 4.0(9) para simplificar a criação de serviços RESTful. Ela combina recursos das anotações @Controller e @ResponseBody, eliminando a necessidade de anotar todos os métodos de tratamento de solicitações da classe *controller* com a anotação @ResponseBody.

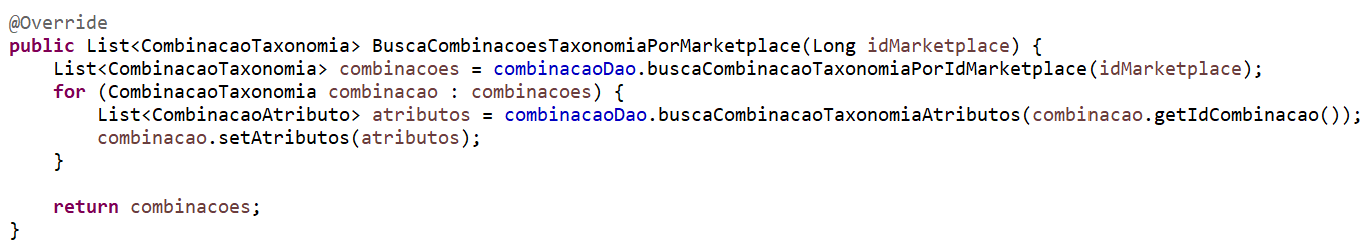
Para o usuário ter acesso a API bastar fazer uma requisição para a URL “/getCombinacoesPorMarketplace” informando o parâmetro “idMarketplace” com o id do *marketplace* ao qual deseja consultar as combinações existentes. No *controller* que mapeia esta URL a aplicação realiza uma busca em todas as combinações atreladas ao id do *marketplace* informado. Assim como em todas as outras funcionalidades do protótipo, a classe da camada *controller* acessa a camada *service*, que acessa a camada de DAO. A Figura 2000 retrata o *controller* mencionado, seguido pela Figura 2100 com o trecho de código da camada *service* com o método chamado pelo *controller.* Por fim, a Figura 2200 detalha o código da camada de DAO utilizada. As consultas no banco de dados realizadas nesta etapa são as mais complexas da aplicação, ligando quase todas as tabelas existentes. Isso faz com que o JSON retornado pela API disponha de todas as informações necessárias.

Figura 2000 – Controller “ApiCombinacaoController”



Fonte: Elaborado pelo autor (2019)

Figura 2100 – Método “BuscaCombinacoesTaxonomiaPorMarketplace”



Fonte: Elaborado pelo autor (2019)

Figura 2100 – Métodos da camada de DAO utilizados



Fonte: Elaborado pelo autor (2019)

TESTES E VALIDAÇÕES

Logo no início do desenvolvimento do protótipo foi acordada a utilização dos dados da plataforma do *marketplace* da empresa XY. Desde então, um analista da empresa se dispôs a realizar quaisquer testes necessários. Conforme o desenvolvimento das funcionalidades era concluído o analista era contatado para realização de testes unitários e individuais. Segundo XXX (12), apesar de básicos, os testes unitários são fundamentais. Este tipo de teste possibilita que todos os módulos do sistema sejam analisados individualmente, verificando cada funcionalidade.

Os eventuais problemas encontrados eram corrigidos e o sistema era testado novamente antes do desenvolvimento das próximas etapas. Após a conclusão do desenvolvimento foi realizado um teste integrado do protótipo, baseando-se nos casos de uso.

Existem diversas abordagens para realização de testes de software em geral. Para os testes do protótipo desenvolvido, foi utilizada a técnica caixa-preta. Conforme explica XXX (24), esta técnica pode ser utilizada para encurtar o tempo de realização dos testes em sistemas altamente complexos, uma vez que não leva em consideração o comportamento interno do sistema. Neste tipo de teste os dados de entrada são fornecidos ao sistema que realiza determinado processo. Então, as saídas decorrentes destas entradas são comparadas com os resultados esperados já documentados. Para auxiliar o analista da empresa XY na realização dos testes, foram criados os casos de teste listados no Apêndice B. O modelo de caso de teste utilizado foi proposto por XXX (2019).

|  |  |
| --- | --- |
| **Caso de teste** | |
| **ID:** | 1 |
| **Nome:** | Gerenciar sistema |
| **Ambiente:** | Navegador Google Chrome |
| **Ator:** | Vendedor e plataforma de *marketplace*. |
| **Pré-condições:** | Não há. |
| **Procedimento:** | 1. Acessar o sistema através da URL. 2. Acessar a opção “Cadastrar Marketplace”. 3. Clicar no botão voltar. 4. Acessar a opção “Cadastrar Vendedor”. 5. Clicar no botão cancelar. 6. Acessar a opção “Criar Nova Combinação”. 7. Clicar no botão cancelar. 8. Acessar a opção “Consultar e Editar Combinações”. |
| **Pós-condições:** | Percorrer por todas as telas sem que nenhum erro seja retornado. |

|  |  |
| --- | --- |
| **Caso de teste** | |
| **ID:** | 2 |
| **Nome:** | Manter vendedor e carregar taxonomia vendedor. |
| **Ambiente:** | Navegador Google Chrome |
| **Ator:** | Vendedor e API vendedor. |
| **Pré-condições:** | Não há. |
| **Procedimento:** | 1. Acessar a opção “Cadastrar Vendedor. 2. Clicar no botão “Adicionar Vendedor”. 3. Preencher os campos. 4. Clicar em “Salvar”. 5. Aguardar até que o cadastro seja salvo. 6. Após o redirecionamento ao menu inicial, clicar em “Adicionar Vendedor”. 7. Informar o nome do vendedor cadastrado no campo “Nome do Vendedor” 8. Clicar em buscas. 9. Acessar o vendedor cadastrado previamente. 10. Editar alguma informação. 11. Clicar em salvar. |
| **Pós-condições:** | Cadastro de vendedor criado com sucesso, busca realizada com sucesso e edição de cadastro realizada com sucesso. |

Coisas para arrumar:

2- Adicionar bibliografia

ENGHOLM JUNIOR, Hélio. **Análise e Design Orientados a Objetos.**São Paulo: Novatec, 2013. 376 p.

3 – Adicionar rodapés: *Thymeleaf*, *bootstrap*, *javascript* e *jquery*.

5 - <https://www.thymeleaf.org/>

6 - <https://getbootstrap.com/>

7- <https://jquery.com/>

8 - <https://www.thymeleaf.org/doc/tutorials/2.1/thymeleafspring.html>

9 - <https://www.baeldung.com/spring-controller-vs-restcontroller>